Problème 1. Calcul d'enveloppe convexe

Notation Pour m et n deux entiers naturels, $[\![m,n]\!]$ désigne l'ensemble des entiers k tels que $m \leq k \leq n$.

Préambule *Ce problème a pour but d'étudier certains aspects de la géométrie algorithmique qui sont fortement utilisés en ingénierie et surtout en imagerie numérique. L'objectif est d'étudier deux algorithmes déterminant l'enveloppe convexe d'un ensemble de n points. Le premier, dit **parcours de Jarvis**, s'exécute en temps $O(nN)$ où N est le nombre de sommets de l'enveloppe convexe. Le second, dit **balayage de Graham**, s'exécute en temps $O(n \log n)$.*

Dans tout le problème, on se place dans un plan euclidien orienté muni d'un repère orthonormé direct (non visible sur les figures de ce problème).

On pourra utiliser les fonctions de la bibliothèque `math` supposée déjà importée.

On rappelle que la complexité (temporelle) d'un algorithme est le nombre d'opérations élémentaires (affectations, comparaisons, opérations arithmétiques) effectuées par un algorithme. Ce nombre s'exprime en fonction de la taille des données.

Un point P du plan muni d'un repère est représenté en Python par une liste de deux valeurs `P = [x,y]` où `x` et `y` sont deux nombres flottants correspondants aux coordonnées cartésiennes du point. Un nuage de points est un ensemble $L = \{P_0, \ldots, P_{n-1}\}$ fini et non vide de points du plan. On le représente en Python par une liste `L` de longueur n, où pour tout entier i dans $[\![0, n-1]\!]$, `L[i]` représente le point P_i.

1.1 Préliminaires

Question 1.1. Calcul de la distance entre deux points du plan La distance euclidienne entre deux points du plan P et Q de coordonnées respectives (x_P, y_P) et (x_Q, y_Q) est donnée par

$$PQ = \sqrt{(x_P - x_Q)^2 + (y_P - y_Q)^2}.$$

(a) Écrire en Python une fonction `distance` prenant en arguments deux points P et Q du plan et renvoyant la valeur de la distance euclidienne entre ces deux points.

Un élève a écrit une fonction qui prend en argument un nuage de points de taille supérieure à 2 et qui détermine la distance minimale entre deux points de ce nuage. Voici le code de son programme en Python :

```
def distance_minimale(L):
    n = len(L)
    minimum = distance(L[0], L[1])
    i,j = 0,0
    while i < n:
        while j < n:
            a = distance(L[i], L[j])
            if a < minimum:
                minimum = a
            j += 1
        i += 1
    return minimum
```

(b) Combien d'appels à la fonction `distance` sont effectués par cette fonction ?

(c) Quelle est la valeur renvoyée par cette fonction `distance_minimale` ?

(d) Corriger le programme de l'élève afin que la fonction `distance_minimale` soit correcte.

(e) Écrire une fonction `distance_maximale` qui prend en argument un nuage de points L et qui renvoie la distance maximale entre deux points du nuage L en effectuant exactement $n(n-1)/2$ appels à la fonction `distance`. La fonction `distance_maximale` renverra également les indices d'un couple de points réalisant le maximum voulu. *Par exemple, si la fonction reçoit en argument un nuage de point L dans lequel la distance P_1P_8 est égale à* 0.9243140331952826*, qui est la distance maximale, l'exécution produira le résultat suivant.*

```
>>> distance_ maximale(L)
(0.9243140331952826, 1, 8)
```

Question 1.2. Recherche du point d'abscisse minimale

(a) Écrire une fonction `point_abs_min` qui prend en paramètre un nuage de points L et qui renvoie l'indice du point de plus petite abscisse parmi les points du nuage de L. *Si plusieurs points ont une abscisse minimale alors la fonction renverra parmi ces points, l'indice du point d'ordonnée minimale.*

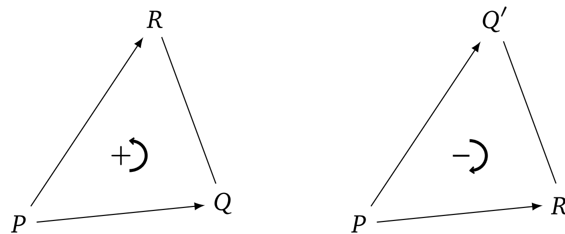
(b) Quelle est la complexité temporelle de votre fonction lorsque le nuage est composé de n points ?

Question 1.3. Détermination de l'orientation de trois points du plan

Définition. Soient P, Q et R trois points du plans. On considère les vecteurs $\vec{PQ}$ et $\vec{PR}$ de coordonnées respectives $\binom{a}{b}$ et $\binom{c}{d}$. On note M la matrice $\begin{pmatrix} a & c \\ b & d \end{pmatrix}$.
On dit que l'orientation du triplet (P,Q,R)

- est *en sens direct* si le déterminant de la matrice M est strictement positif ;
- est *en sens indirect* si le déterminant de la matrice M est strictement négatif ;
- est *un alignement* si le déterminant de la matrice M est nul.

Sur la figure ci-dessous, le triplet (P,Q,R) est en sens direct et le triplet (P',Q',R') est en sens indirect.



(a) On suppose qu'un triplet (P,Q,R) est en sens direct. Quelle est l'orientation des triplets (Q,R,P) et (P,R,Q) ? *Justifier votre réponse.*

(b) Écrire une fonction `orientation` qui prend en arguments 3 points du plan P, Q et R et qui renvoie `1` si le triplet (P,Q,R) est en sens direct, `0` si le triplet (P,Q,R) est un alignement, et `-1` si le triplet (P,Q,R) est en sens indirect.

Question 1.4. Un premier algorithme de tri La fonction `tri_bulle` ci-dessous prend en argument une liste L de nombres flottants et en effectue le tri en ordre croissant.

```
def tri_bulle(L):
    n = len(L)
    for i in range(n):
        for j in range(n-1, i, -1):
            if L[j] < L[j-1]:
                L[j], L[j-1] = L[j-1], L[j]  # échange d'éléments
```

(a) Lors de l'appel `tri_bulle(L)` où L est la liste `[5,2,3,1,4]`, donner le contenu de la liste L à la fin de chaque itération de la boucle `for i in range(n):`.

(b) On suppose que L est une liste non vide de nombres flottants. Montrer, pour tout $k \in [\![0,n]\!]$, la propriété $\mathscr{P}_k$: « après k itérations de la première boucle, les k premiers éléments de la liste sont triés par ordre croissant et sont tous inférieurs aux $n-k$ éléments restants ».

(c) En déduire que `tri_bulle(L)` trie bien la liste L en ordre croissant.

(d) Donner la complexité dans le meilleur des cas et dans le pire des cas de la fonction `tri_bulle`.

Question 1.5. Un deuxième algorithme de tri Soit la fonction `tri_fusion` suivante.

```
def tri_fusion(L):
    """
    Fonction qui prend en argument une liste L de nombres flottants
    et qui trie cette liste
    """
    n = len(L)
    if n <= 1: return L
    m = n//2
    return fusion(tri_fusion(L[0:m]), tri_fusion(L[m:n]))
```

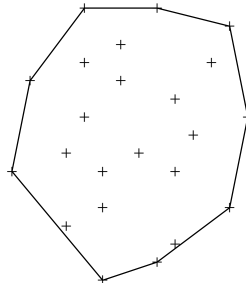
(a) Écrire une fonction `fusion` qui prend en arguments deux listes triées `L1` et `L2` et qui renvoie une seule liste triée contenant les éléments de `L1` et `L2`. La fonction `fusion` devra avoir une complexité en $O(n_1 + n_2)$ où n_1 et n_2 sont les tailles respectives des listes `L1` et de `L2`. *Par exemple l'appel* `fusion([2,4,7],[3,5,6,9])` *renverra la liste* `[2,3,4,5,6,7,9]`.

(b) On admet que la fonction `fusion` de la question précédente se termine. Montrer que la fonction `tri_fusion` se termine également.

(c) On suppose que la longueur de la liste `L` est $n = 2^p$, où p est un entier naturel. Quelles sont alors les complexités dans le meilleur des cas et dans le pire des cas de `tri_fusion` ?

1.2 Enveloppe convexe d'un nuage de points

Dans cette partie, l'objet est d'étudier deux algorithmes permettant d'obtenir l'enveloppe convexe d'un ensemble fini de points du plan.
Définition. Un ensemble S est **convexe** si pour tout couple de points (P, Q) de S, le segment $[PQ]$ est contenu dans S. L'**enveloppe convexe** d'un ensemble S est le plus petit ensemble convexe contenant S.
Théorème (admis). Soit S un ensemble de n points du plan, avec $n > 1$. l'enveloppe convexe de S est constituée par un polygone P, sous-ensemble de S, des segments unissant les points successifs de P, et de tous les segments unissant deux points des segments précédents, comme l'illustre la figure ci-dessous.



Dans la suite du problème, on supposera que les nuages de points considérés ne contiennent pas 3 points distincts alignés. Cette hypothèse permettra de simplifier les algorithmes.

Question 1.6. La « **marche de Jarvis** » est un des algorithmes les plus naturels. Conçu en 1973, il consiste à reprendre l'image de l'emballage d'un cadeau. La première idée de Jarvis pour déterminer l'enveloppe convexe est de chercher tout d'abord les segments qui forment ses côtés à l'aide de la propriété admise suivante.
Propriété. Soit L un nuage de points. Pour tous points distincts P et Q de L, le segment $[PQ]$ est un côté de l'enveloppe convexe de L si tous les triplets (P, Q, R) où R est un point de L distinct de P et Q ont la même orientation.
On en déduit une fonction « naïve » qui calcule les sommets de l'enveloppe convexe d'un nuage de points.

```
1  def jarvis(L):
2      """
```

```
Fonction qui reçoit en argument un nuage de points et qui renvoie
une liste contenant les indices des sommets de l'enveloppe
convexe de ce nuage
"""
    n = len(L)
    EnvConvexe = []
    for i in range(n):
        for j in range(n):
            Listeorientation = []
            if i != j:
                for k in range(n):
                    if k != i and k != j:
                        Listeorientation.append(orientation(L[i], L[j], L[k]))
                a = Listeorientation[0]
                sommet = True
                for v in Listeorientation:
                    if v != a:
                        sommet = False
                if sommet and i not in EnvConvexe:
                    EnvConvexe.append(i)
                if sommet and j not in EnvConvexe:
                    EnvConvexe.append(j)
    return EnvConvexe
```

(a) Expliquer à quoi servent les lignes 16 à 20 de cette fonction.

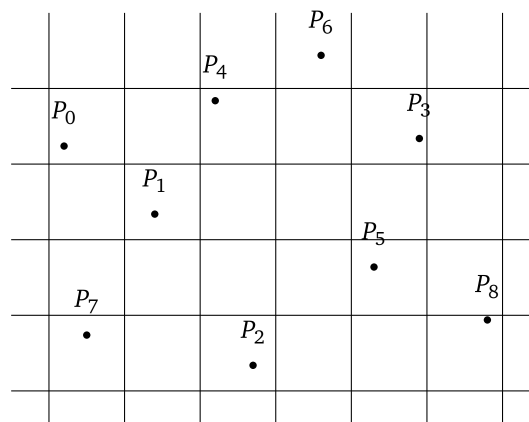
(b) Si on considère un nuage de n points avec $n > 3$, quelle est la complexité temporelle de cette fonction ? *Justifier.*

(c) Le script suivant trace-t-il le polygone formé par les sommets de l'enveloppe convexe du nuage L ? *Justifier.*

```
>>> import matplotlib.pyplot as plt
>>> Enveloppe = jarvis(L)
>>> plt.plot([L[i][0] for i in Enveloppe], [L[i][1] for i in Enveloppe])
>>> plt.show()
```

Question 1.7. Jarvis proposa ensuite une version plus efficace de son algorithme en voulant « ranger » les points du nuage autour d'un point de l'enveloppe. Il utilise une relation d'ordre sur les points du nuage.
Définition. Soit P un sommet de l'enveloppe convexe du nuage. Le point suivant sur l'enveloppe convexe est le plus petit pour la relation d'ordre $\preccurlyeq_P$ définie sur l'ensemble des sommets du nuage différents de P par

$$Q \preccurlyeq_P R \iff \text{le triplet } (P,Q,R) \text{ est en sens direct ou } Q = R.$$



Par exemple, dans le nuage points ci-dessus, on obtient le classement des points suivants pour la relation d'ordre $\preccurlyeq_{P_2}$:

$$P_8 \preccurlyeq_{P_2} P_5 \preccurlyeq_{P_2} P_3 \preccurlyeq_{P_2} P_6 \preccurlyeq_{P_2} P_4 \preccurlyeq_{P_2} P_1 \preccurlyeq_{P_2} P_0 \preccurlyeq_{P_2} P_7.$$

Le plus petit point pour la relation $\preccurlyeq_{P_2}$ est le point P_8.

(a) Donner le classement des points du nuage de la figure ci-dessus pour la relation d'ordre $\preccurlyeq_{P_6}$.

Ainsi si on connaît un sommet P de l'enveloppe convexe d'un nuage de points alors le prochain point de l'enveloppe convexe à déterminer est celui qui est minimal pour la relation $\preccurlyeq_P$.

(b) Écrire une fonction `prochain_sommet` qui prend en arguments un nuage de points L et l'indice d'un sommet P de ce nuage de points qui est un sommet de l'enveloppe convexe et qui renvoie l'indice du prochain sommet de l'enveloppe convexe. Cette fonction devra avoir une complexité en $O(n)$ où n est le nombre de points du nuage.

Comme le point du nuage d'abscisse minimale (et d'ordonnée minimale s'il y a plusieurs points d'abscisse minimale) fait partie de l'enveloppe convexe on peut construire un algorithme qui détermine au fur et à mesure tous les sommets de l'enveloppe convexe. On arrête l'algorithme quand la fonction `prochain_sommet` renvoie l'indice du sommet de départ.

(c) Recopier et compléter la fonction suivante afin qu'elle renvoie l'enveloppe convexe d'un nuage de points.

```
def jarvis2(L):
    i = point_abs_min(L)
    suivant = prochain_sommet(L, i)
    # initialisation de la liste des sommets de l'enveloppe convexe :
    Enveloppe = [i, suivant]
    while ......:
        ......
        ......
    return Enveloppe
```

(d) Soit L un nuage de n points dont on sait que l'enveloppe convexe est un polygone à N sommets. Montrer que l'algorithme décrit par la fonction `jarvis2` possède une complexité en $O(n \times N)$.

Question 1.8. L'algorithme de Graham - Andrew En 1972, Graham et Andrew proposèrent une méthode pour déterminer l'enveloppe convexe d'un nuage de points. Leur algorithme est basé sur une méthode de tri des points.

La première étape de l'algorithme de Graham et Andrew est de trier les n points du nuage L par ordre croissant d'abscisses (si deux points ont la même abscisse on classera ces points suivant leurs ordonnées). On supposera donnée une fonction `tri_nuage` prenant en entrée un nuage de points L et réalisant cette opération en complexité $O(n \log n)$ où n est le nombre de points du nuage ($n \geq 3$).

L'idée de l'algorithme est de balayer le nuage de points dans l'ordre croissant de leurs abscisses tout en mettant à jour l'enveloppe convexe des points. L'enveloppe convexe a été scindée en deux listes `EnvSup` et `EnvInf`. Dans ces listes `EnvSup` et `EnvInf`, on maintient l'enveloppe convexe des points déjà traités. Chaque nouvel indice du point P du nuage est ajouté à `EnvSup` et `EnvInf`, puis tant que l'avant dernier-point de `EnvSup` rend la séquence non-convexe, il est enlevé (de même pour `EnvInf`).

Voici en Python la fonction qui permet d'obtenir l'enveloppe convexe :

```
def graham_andrew(L):
    L = tri_nuage(L)
    EnvSup =[]
    EnvInf =[]
    for i in range(len(L)):
        while len(EnvSup) > 1 and \
              orientation(L[i], L[EnvSup[-1]], L[EnvSup[-2]]) <= 0:
            EnvSup.pop()
        EnvSup.append(i)
```

```
        while len(EnvInf) > 1 and \
                orientation(L[EnvInf[-2]], L[EnvInf[-1]], L[i]) <= 0:
            EnvInf.pop()
        EnvInf.append(i)
    return EnvInf[:-1] + EnvSup[::-1]
```

(a) À partir du nuage de points représentés à la question 1.7, donner le contenu de la liste `EnvSup` à chaque itération de la boucle `for`. Donner également le résultat de `orientation(P, EnvSup[-1], EnvSup[-2]) <= 0`.

(b) Montrer la terminaison de la fonction `graham_andrew`.

(c) Montrer que la complexité de `graham_andrew` est en $O(n \log n)$ où n est la taille du nuage de points.